LSTMs and GRU-RNNs

AW

Lecture Overview

1. LSTM
2. GRU-RNNs
3. Clipping Gradients
4. Batch Normalization

Gated RNNs

- Idea - create paths through time that have derivatives that neither vanish nor explode (same as with leaky units and skip connections)
- Implementation: generalize the idea of leaky units – allow weights change in time with each time step

Gated units vs leaky units:

- Leaky units accumulate the information over given period of time
- Gated units may accumulate and forget the information

Example:

- A sequence consist of subsequences. We need to remember the information inside subsequence and forget info outside of it.
- Can set it manually by the length on subsequence if known
- If not known can try to learn it as RNN parameter

LSTM: Long Short-Term Memory

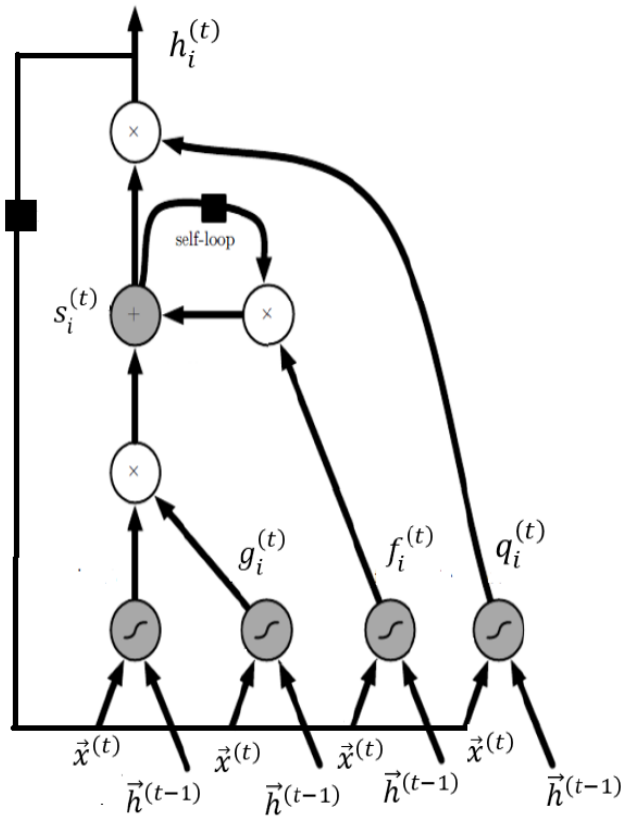
- Key contribution: introducing self-loops to produce paths where the gradient can flow for a long time
  - make the weight on this self-loop conditioned on the context, rather than fixed
- Implementation: make the weight of self-loop controlled by another hidden unit (gated).
  - This means that the time scale of integration/accumulation is changing dynamically dependent on another hidden unit.

One cell of LSTM with 'forget' Gates (1 of 5)

Cells are connected recurrently to each other. One LSTM unit is used instead of 1 hidden unit in RNN

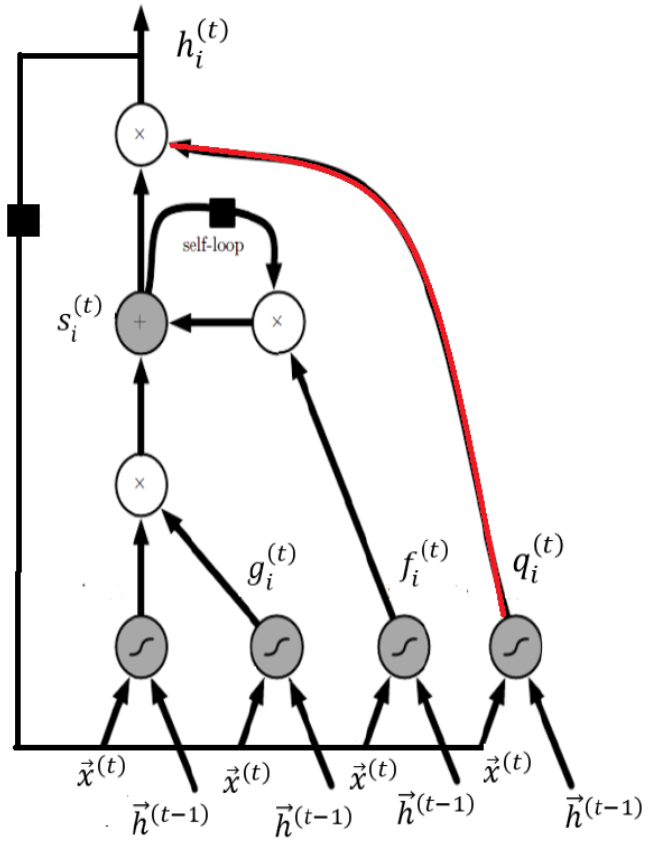
- Input is given by a regular neuron unit (sigmoid).
- Input weight (shutting off) is controlled by sigmoidal input gate.
- The allowed input is accumulated into the state
- The state unit has a linear self-loop
- Weight of self loop is controlled (shut-off) by the forget gate
- The output of the cell is controlled (can be shut off) by the output gate

All the gating units are sigmoid. Input unit can have any nonlinearity



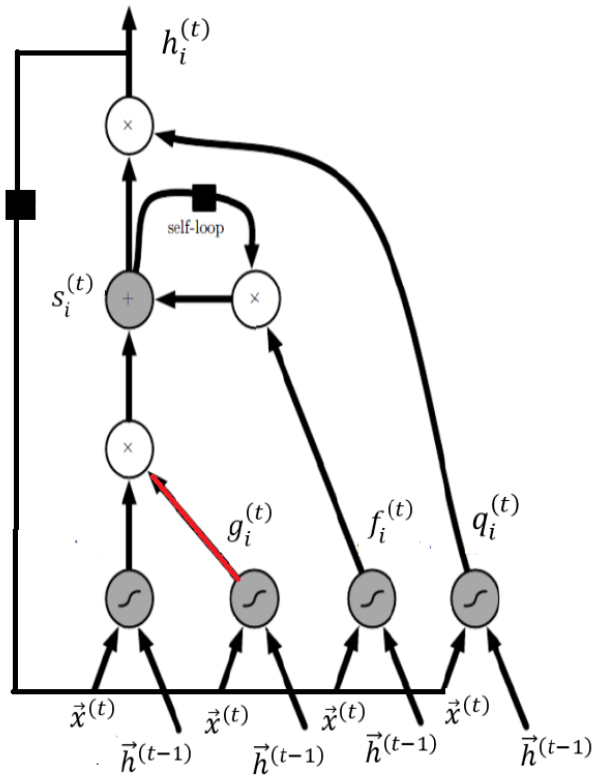
One cell of LSTM with 'forget' Gates (2 of 5)

- The output step is to decide what information we're going to throw away from the cell state. This decision is made by a sigmoid layer called the "output gate layer" denoted by $q_i^{(t)}$ It looks at $\vec{h}_{t-1}$ and $\vec{x}_t$, and outputs a number between 0 and 1 for the state. A 1 represents "completely keep this" while a 0 represents "completely get rid of this."



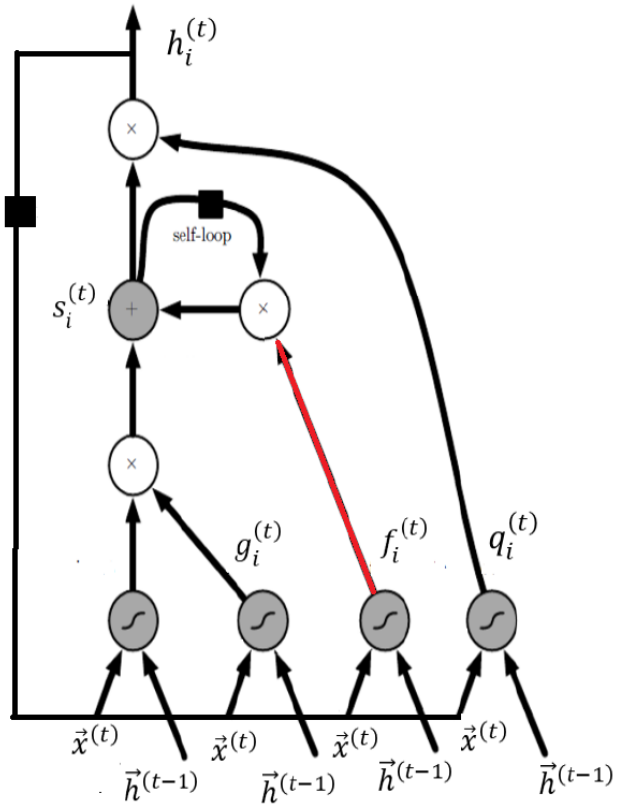
One cell of LSTM with 'forget' Gates (3 of 5)

- The internal step is to decide what new information we're going to store in the cell state. This has three parts.
  1. First, a sigmoid layer called the "external input gate layer" $\vec{g}_i$ is a vector between 0 and 1 that for a given $\vec{h}_{t-1}$ and $\vec{x}_t$ decides which values of input and to what extent to keep



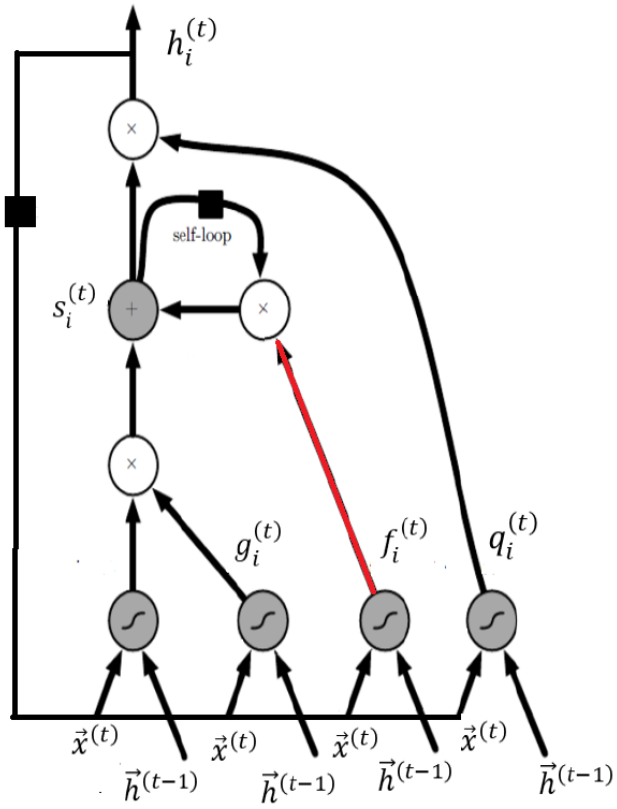
One cell of LSTM with 'forget' Gates (4 of 5)

- The internal step is to decide what new information we're going to store in the cell state. This has three parts.
  2. Next, a sigmoid layer called the "forget gate layer" produce value f_i between 0 and 1 that decides for a give $\vec{h}_{t-1}$ and $\vec{x}_t$ what fraction of the state (memory) is contributing to the next state value



One cell of LSTM with 'forget' Gates (5 of 5)

- The internal step is to decide what new information we're going to store in the cell state. This has three parts.
  3. Finally based on the allocated portions of input and previous state a $\tanh$ layer creates a new state variable value $s_i^{(t)}$ that is also a 'candidate for output' value.
- These candidate for output values produce output $h_i^{(t)}$ values in accordance with the fraction allocated by forget gate

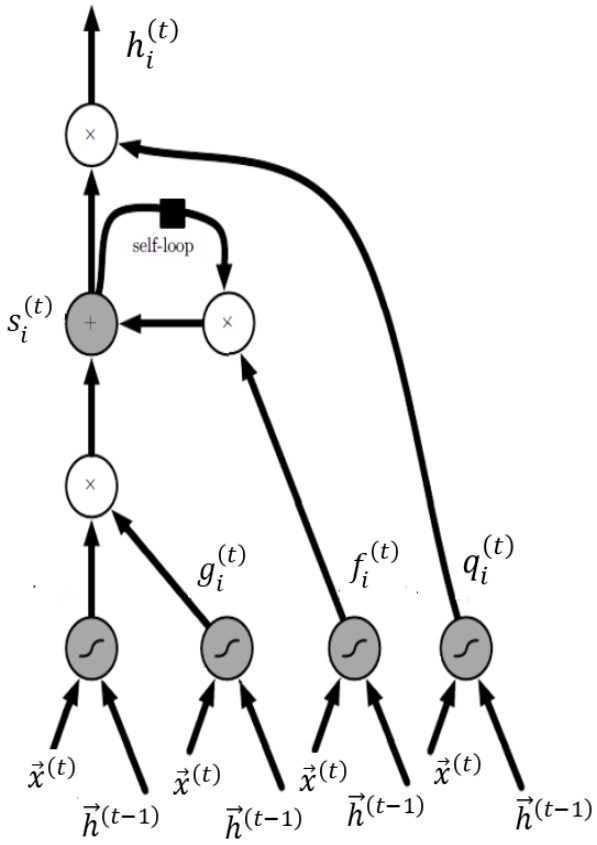


One LSTM Cell Computationally (1 of 4)

- In time slice t the i^{th} state unit is denoted $s_i^{(t)}$. It has a linear self-loop like the leaky units
- Self-loop weight is controlled by a forget gate unit $f_i^{(t)}$ that sets this weight to a value between 0 and 1 via a sigmoid unit:

$$f_i^{(t)} = \sigma\left(b_i^f + \sum_j u_{ij}^f x_j^{(t)} + \sum_j w_{ij}^f h_j^{(t-1)}\right)$$

where $\vec{b}^f$ is a bias vector of forget gate, $U^{(f)} = [u_{ij}^f]$ are the input-to-forget weights, $\vec{x}^{(t)}$ is input vector, $W^f = [w_{ij}^f]$ are hidden-to-forget weights and $\vec{h}^{(t-1)}$ are outputs of previous time slice LSTMs.

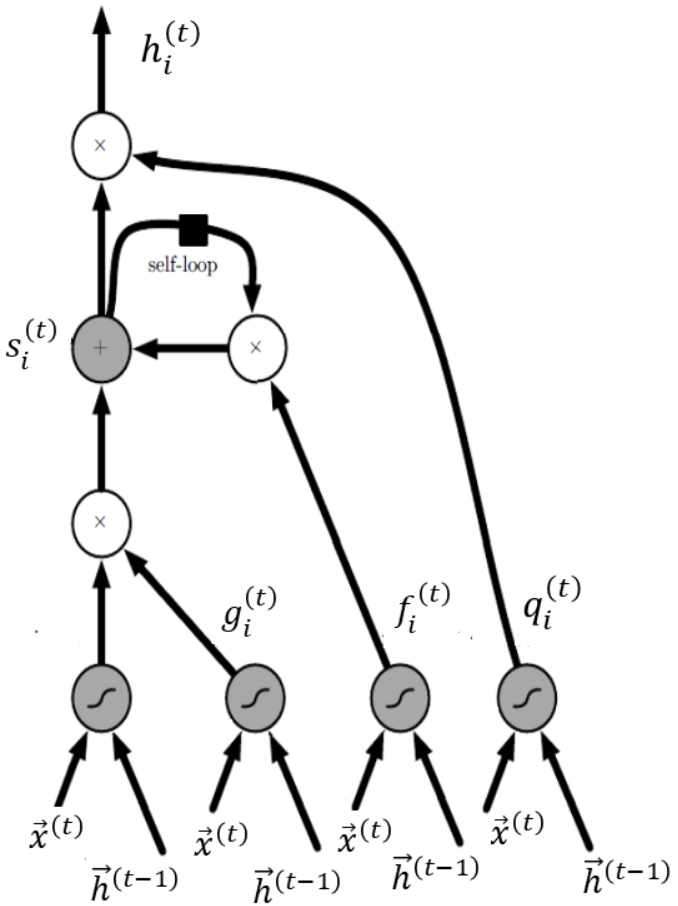


One LSTM Cell Computationally (2 of 4)

- The external input gate unit $g_i^{(t)}$ is computed similarly to the forget gate with a sigmoid unit to obtain a gating value between 0 and 1:

$$g_i^{(t)} = \sigma\left(b_i^g + \sum_j u_{ij}^g x_j^{(t)} + \sum_j w_{ij}^g h_j^{(t-1)}\right)$$

where $\vec{b}^g$ is a bias vector of external input gate, U^g are the input-to-external input gate weights, $\vec{x}^{(t)}$ is input vector, W^f are hidden-to-external input gate weights and $\vec{h}^{(t-1)}$ are outputs of previous time slice LSTMs

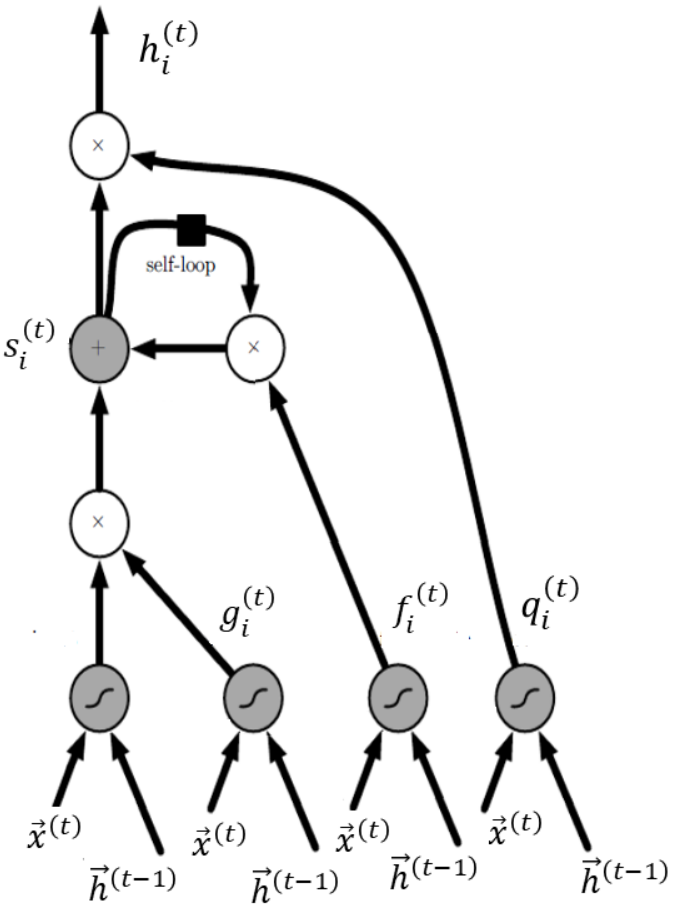


One LSTM Cell Computationally (3 of 4)

- State variable of gate i is computed as

$$s_i^{(t)} = f_i^{(t)} s_i^{(t-1)} + g_i^{(t)} \sigma\left(b_i + \sum_j u_{ij} x_j^{(t)} + \sum_j w_{ij} h_j^{(t-1)}\right)$$

where contribution $s_i^{(t-1)}$ of its own state in previous time slice is controlled by value of forget gate, and contribution of input $\vec{x}^{(t)}$ and hidden output $\vec{h}^{(t-1)}$ is controlled by value of input gate $\vec{g}^{(t)}$, $\vec{b}$ is the state bias, U are input-to-state weights, W are hidden-to-input weights



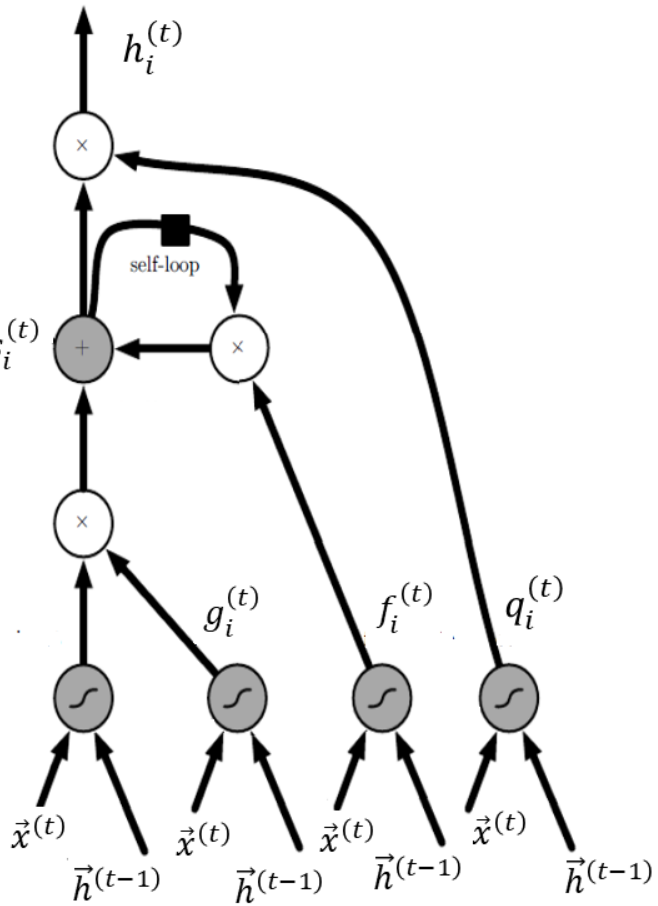
One LSTM Cell Computationally (4 of 4)

- The output is controlled by output gate unit $q_i^{(t)}$ that is computed similarly to the forget and external input gate but with its own parameters:

$$q_i^{(t)} = \sigma\left(b_i^0 + \sum_j u_{ij}^o x_j^{(t)} + \sum_j w_{ij}^o h_j^{(t-1)}\right)$$

- It controls out put of hidden LSTM unit that is computed as

$$h_i^{(t)} = q_i^{(t)} \tanh\left(s_i^{(t)}\right)$$



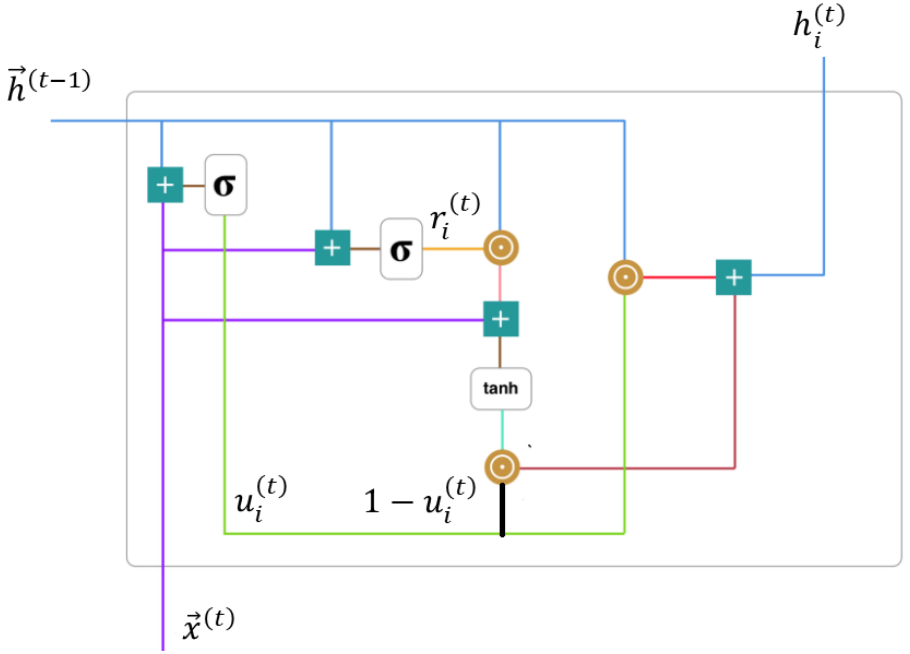
Note: In this LSTM we do not have delayed input of state to control gates. But we could. This scheme is computationally more expensive but even more stable

Lecture Overview

1. LSTM
2. GRU-RNNs
3. Clipping Gradients
4. Batch Normalization

Gated Recurrent Units (GRUs)

- Simplified LSTMs
- The main difference is that a single gating unit simultaneously controls the forgetting factor in self state-updates and input factors in state-updates
- No explicit state is necessary as the memory is provided by linearly combining input and short term memory. It consist of
  1. Update gate
  2. Reset gate
  3. Current memory

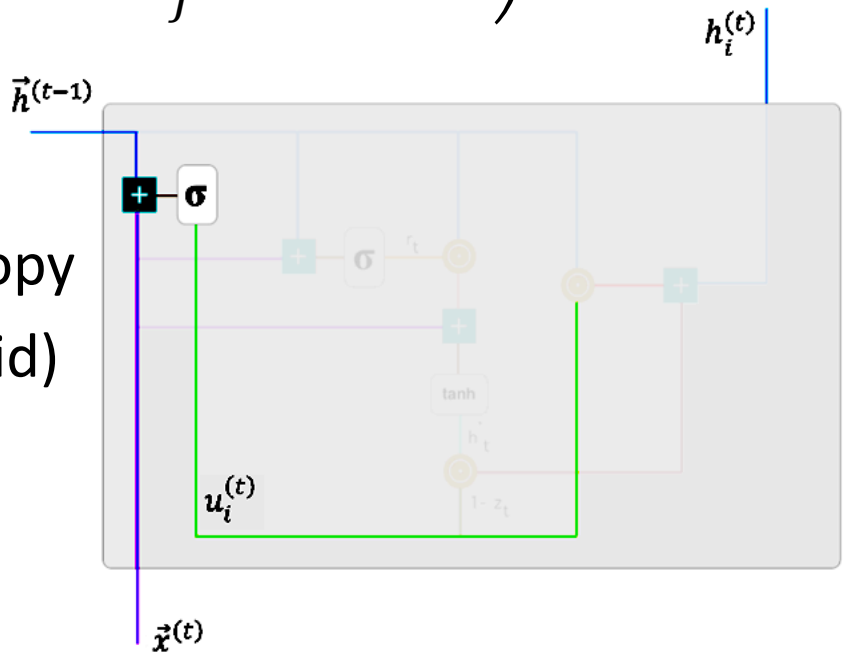


Update Gate of GRU

- The update gate helps the model to determine how much of the past information (from previous time steps) needs to be passed along to the future. It is defined as:

$$u_i^{(t)} = \sigma\left(b_i^u + \sum_j U_{ij}^u x_j^{(t)} + \sum_j W_{ij}^u h_j^{(t-1)}\right)$$

- It acts as conditional leaky Integrator that can linearly gate memory by choosing to copy it (at one extreme of the sigmoid) or completely ignore it (at the other extreme) – see current memory unit ahead

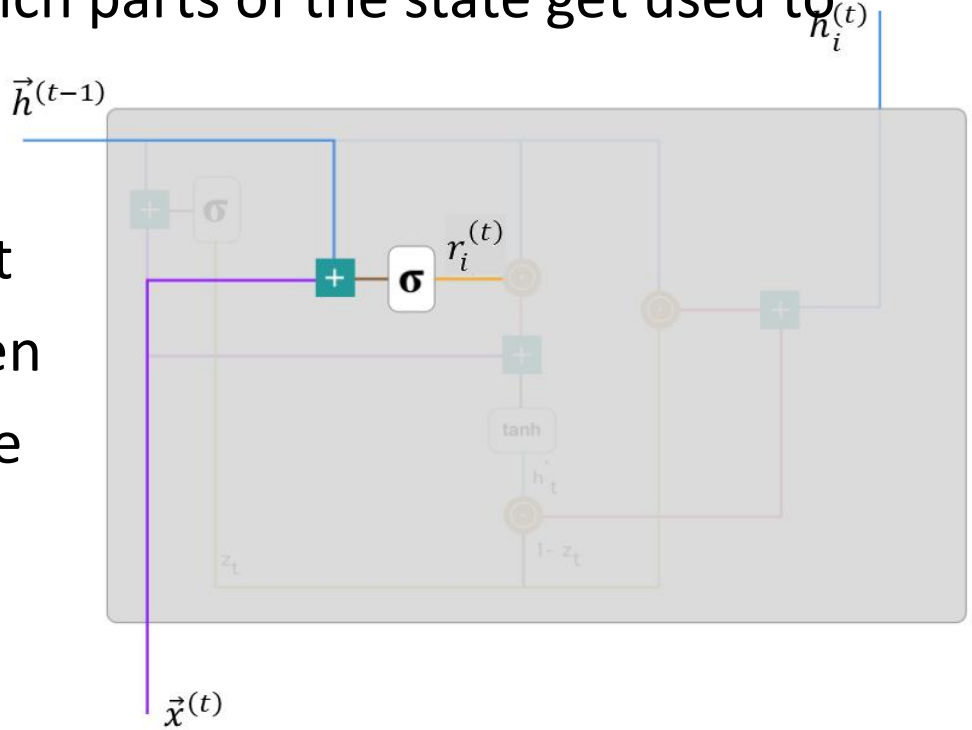


Reset Gate of GRU

- Reset gate is used from the model to decide how much of the past information to forget. It is defined as:

$$r_i^{(t)} = \sigma\left(b_i^r + \sum_j U_{ij}^r x_j^{(t)} + \sum_j W_{ij}^r h_j^{(t-1)}\right)$$

- The reset gates control which parts of the state get used to compute the next target state, introducing an additional nonlinear effect in the relationship between past state and future state

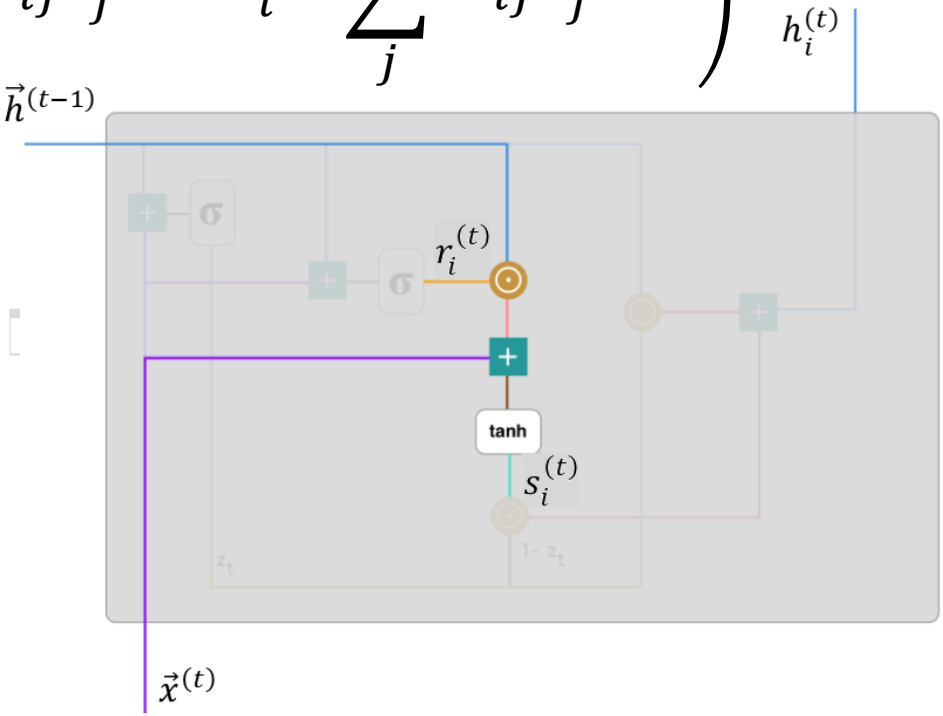


Current Memory State

- There is no explicit gate for it but it is determined by linear combination of two terms: one term is the product of reset gate value and hidden output in previous times slice and another term is current input:

$$s_i^{(t)} = \tanh\left(b_i + \sum_j U_{ij} x_j^{(t)} + r_i^{(t)} \sum_j W_{ij} h_j^{(t-1)}\right)$$

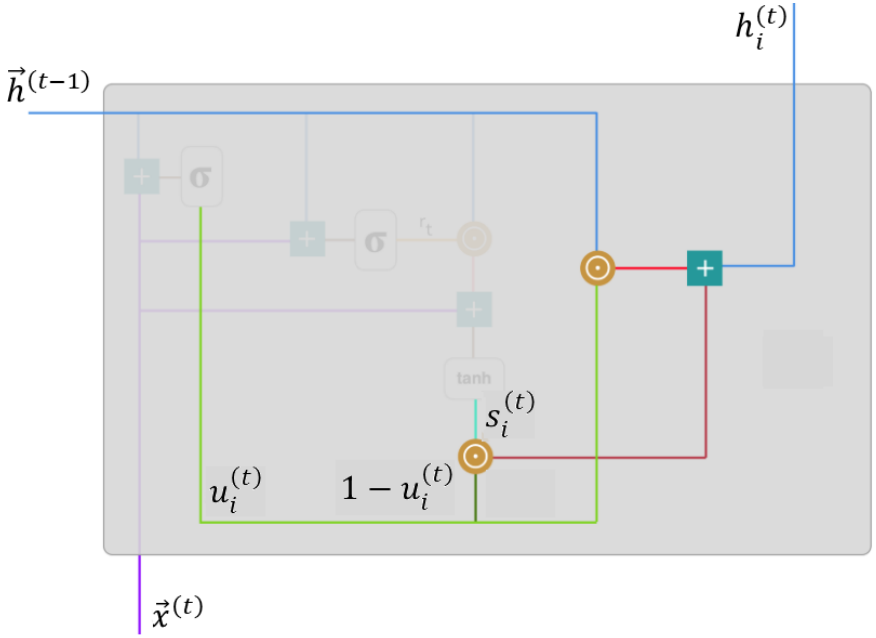
- In GRU current memory state isn't explicitly dependent on its previous memory state, so it is not explicitly computed inside the unit



The Output of Hidden GRU

- Finally GRU is ready to calculate $\vec{h}^{(t)}$ - vector which holds information for the current unit and passes it down to the network. For that the update gate $u_i^{(t)}$, current state $s_i^{(t)}$, and old time slice hidden output are needed:

$$h_i^{(t)} = u_i^{(t)} h_i^{(t-1)} + \left(1 - u_i^{(t)}\right) s_i^{(t)}$$



Lecture Overview

1. LSTM
2. GRU-RNNs
3. Clipping Gradients
4. Batch Normalization

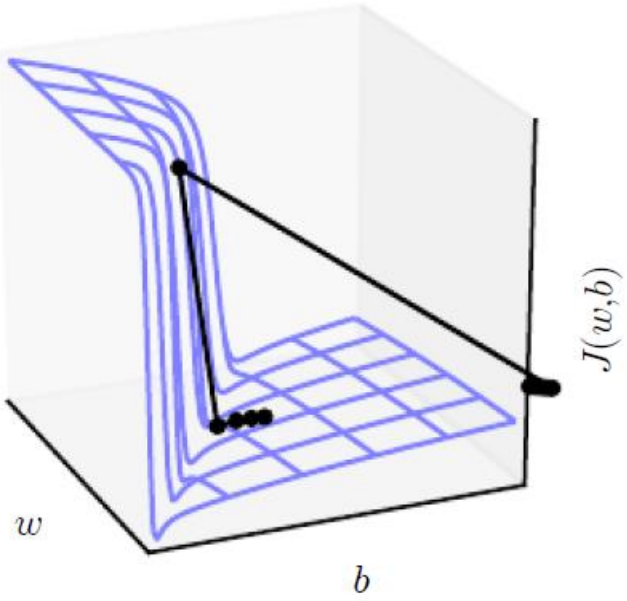
Exploding Gradient

- strongly nonlinear functions (e.g. σ or $\tanh$ computed in RNNs over many time steps) tend to have derivatives that can be either very large or very small
- The gradient gives the direction of the steepest descent within *an infinitesimal region surrounding the current parameters*. Outside of the infinitesimal region, the cost function may begin to curve back upwards. The update must be chosen to be small enough to avoid jumping too much upward curvature
- But when the objective function (as a function of the parameters) has a "landscape" in which there are wide and rather flat regions separated by tiny regions where the objective function changes quickly, forming a kind of cliff gradient throws net out of the region

Exploding Gradient: Example

Example:

- "Cliff" behavior of objective function of Two parameters: w and b
  - Gradient descent overshoots the bottom of this small ravine, then receives a very large gradient from the cliff face. The large gradient catastrophically propels the parameters outside the axes of the plot.



- Good learning rates are normally are set to decay slowly enough for consecutive steps to have approximately the same learning rate
- A step size that is appropriate for a relatively linear part of the landscape is inappropriate and causes uphill motion when curved part is entered

Clipping Gradient

Idea: to fight cliffs regularize the gradient by its norm by clipping it – i.e. if the gradient magnitude is overshooting some limit then cap its size to max-vector in the right direction just before the parameter update.

- To implement do

$$if\ \|\vec{g}\| > v\ then\ g \leftarrow \frac{v\vec{g}}{\|g\|}$$

Because the gradient of all the parameters (including different groups such as weights and biases) is renormalized jointly with a single scaling factor, it guarantees that each step is still in the gradient direction

Lecture Overview

1. LSTM
2. GRU-RNNs
3. Clipping Gradients
4. Batch Normalization

Vanishing/Exploding Gradient Problems Again

Consider RNN with one node per time slice and fate of one input

- Forward propagation multiplicatively depends on each weight and activation function evaluation.
- Backpropagated partial derivative get multiplied by weights and activation function derivatives.
- Unless the values are exactly one, the partial derivatives will either continuously increase (explode) or decrease (vanish).
- Hard to initialize weights exactly right.
- Batch normalization fights both vanishing and exploding gradient since it ensures (somewhat) more stable inputs to each layer.

Batch Normalization – Add a Layer

- Add an additional layer than normalizes in *batch-wise* fashion
- Can normalize either
  - Output after a unit (post-activation)
  - Activation inputs before taking linear combinations (pre-activation)
- Additional learnable parameters to ensure that optimal level of nonlinearity is used.
- Pre-activation normalization more common than post-activation normalization.

(a) Post-activation normalization (b) Pre-activation normalization

Batch Normalization (cont.)

i^{th} unit contains two parameters β_i and γ_i that need to be learned.

- Normalize over *batch* of m instances for i^{th} unit. Here $v_i^{(r)}$ is activation value for r^{th} instance before the normalization
- $\mu_i = \frac{\sum_{k=1}^{m} v_i^{(k)}}{m}$ [Batch Mean]
- $\sigma_i^2 = \sum_{k=1}^{m} \frac{\left(v_i^{(k)} - \mu_i\right)^2}{m}$ [Batch Variance]
- $\hat{v}_i^{(r)} = \frac{v_i^{(r)} - \mu_i}{\sigma_i}$ [Normalize Batch Instances]
- $a_i^{(r)} = \gamma_i \cdot \hat{v}_i^{(r)} + \beta_i$ [Scale with Learnable Parameters]

  where $a_i^{(r)}$ are new activation instances
- Why do we need β_i and γ_i? Why find $\hat{v}_i^{(r)}$ only to allow non-0 mean, non-standardized variance again?
- The mean/variance of the batch were determined by a complicated interaction between the parameters in the layers below this unit
- The mean/variance in new activation are determined by β_i / γ_i only which is much easier to learn with gradient decent

Changes to Backpropagation

- We need to backpropagate through the newly added layer of normalization nodes.
  - The batch normalization node can be treated like any other node: we backpropagate through computation of mean and variance.
  - This means that he gradient will never propose an operation that acts simply to increase the standard deviation or mean of activation!
- We want to optimize the parameters β_i and γ_i
  - The gradients with respect to these parameters are computed during backpropagation.
- We need to compute $\frac{\partial L}{\partial \beta_i}, \frac{\partial L}{\partial \gamma_i}$ for parameter updates and $\frac{\partial L}{\partial v_i^{(r)}}$ to send it backwards. Only $\frac{\partial L}{\partial v_i^{(r)}}$ is non-trivial:
- $\frac{\partial L}{\partial \beta_i} = \sum_{r=1}^{m} \frac{\partial L}{\partial a_i^{(r)}} \cdot \frac{\partial a_i^{(r)}}{\partial \beta_i} = \sum_{r=1}^{m} \frac{\partial L}{\partial a_i^{(r)}}$
- $\frac{\partial L}{\partial \gamma_i} = \sum_{r=1}^{m} \frac{\partial L}{\partial a_i^{(r)}} \cdot \frac{\partial a_i^{(r)}}{\partial \gamma_i} = \sum_{r=1}^{m} \frac{\partial L}{\partial a_i^{(r)}} \cdot \hat{v}_i^{(r)}$

Changes to Backpropagation (cont.)

$$\frac{\partial L}{\partial v_i^{(r)}} = \frac{\partial L}{\partial a_i^{(r)}} \cdot \frac{\partial a_i^{(r)}}{\partial v_i^{(r)}} + \frac{\partial L}{\partial \mu_i} \cdot \frac{\partial \mu_i}{\partial v_i^{(r)}} + \frac{\partial L}{\partial \sigma_i} \cdot \frac{\partial \sigma_i}{\partial v_i^{(r)}} = \frac{\partial L}{\partial a_i^{(r)}} \cdot \frac{\partial a_i^{(r)}}{\partial v_i^{(r)}} + \frac{1}{m}\frac{\partial L}{\partial \mu_i} + \frac{(v_i^{(r)} - \mu_i)}{m\sigma_i} \cdot \frac{\partial L}{\partial \sigma_i}$$

where

$$\frac{\partial L}{\partial a_i^{(r)}} \cdot \frac{\partial a_i^{(r)}}{\partial \hat{v}_i^{(r)}} \cdot \frac{\partial \hat{v}_i^{(r)}}{\partial v_i^{(r)}} = \frac{\gamma_i}{\sigma_i} \cdot \frac{\partial L}{\partial a_i^{(r)}} \qquad (1)$$

$$\frac{\partial L}{\partial \mu_i} = \sum_{k=1}^{m} \frac{\partial L}{\partial \hat{v}_i^{(k)}} \frac{\partial \hat{v}_i^{(k)}}{\partial \mu_i} = \sum_{k=1}^{m} \frac{\partial L}{\partial \hat{v}_i^{(k)}} \cdot \left(-\frac{1}{\sigma_i} - \frac{(v_i^{(k)} - \mu_i) \sum_{s=1}^{m} \left(v_i^{(s)} - \mu_i \right)}{m\sigma_i^3} \right) = -\frac{\gamma_i}{\sigma_i} \sum_{k=1}^{m} \frac{\partial L}{\partial a_i^{(k)}}$$

so

$$\frac{\partial L}{\partial a_i^{(r)}} \cdot \frac{\partial a_i^{(r)}}{\partial \hat{v}_i^{(r)}} \cdot \frac{\partial \hat{v}_i^{(r)}}{\partial \mu_i} \cdot \frac{\partial \mu_i}{\partial v_i^{(r)}} = -\frac{1}{m} \cdot \frac{\gamma_i}{\sigma_i} \cdot \sum_{k=1}^{m} \frac{\partial L}{\partial a_i^{(k)}} \qquad (2)$$

and

$$\frac{\partial L}{\partial \sigma_i} = \sum_{k=1}^{m} \frac{\partial L}{\partial \hat{v}_i^{(k)}} \frac{\partial \hat{v}_i^{(k)}}{\partial \sigma_i} = \sum_{k=1}^{m} \frac{\partial L}{\partial a_i^{(k)}} \cdot \gamma_i \cdot \left(-\frac{v_i^{(k)} - \mu_i}{\sigma_i^2} \right) = \frac{\gamma_i}{\sigma_i} \cdot \left(\sum_{k=1}^{m} \frac{\partial L}{\partial a_i^{(k)}} \cdot \hat{v}_i^{(k)} \right), \text{ so}$$

$$\frac{(v_i^{(r)} - \mu_i)}{m\sigma_i} \cdot \frac{\partial L}{\partial \sigma_i} = \frac{\gamma_i}{m\sigma_i} \cdot \hat{v}_i^{(r)} \cdot \left(\sum_{k=1}^{m} \frac{\partial L}{\partial a_i^{(k)}} \cdot \hat{v}_i^{(k)} \right) \qquad (3)$$

$$\frac{\partial L}{\partial v_i^{(r)}} = (1) + (2) + (3)$$

Reaing

- Ch. 7.5, 7.6, 3.6